A Top-Down Model of an AOP Weaving Process

Sergei Kojarski David H. Lorenz
Northeastern University
Boston, Massachusetts 02115 USA
{kojarski,lorenz}@ccs.neu.edu

ABSTRACT

A novice's mistake in understanding what makes a language AOP is to equate AOP with a specific implementation of AOP, like AspectJ [5, 3]. Another misunderstanding is to equate AOP with SOC in general, rather than realizing that AOP is just a specific tool, amongst many others, for SOC.

Current AOP models seem to have erred in a similar way. Models of AOP either focus on a selected instance of AOP (e.g., pointcut and advice in AspectJ) or generalize over several so-declared AOP instances. Since there is no agreed upon definition of what constitutes AOP and since SOC tools are declared, rather than determined, to be AOP, models derived in this manner are either too specific or overgeneralized.

This position paper presents a novel approach to modeling AOP based on an abstract weaving process. We discuss how a precise weaving process enables to construct AOP mechanisms, and supports the composition of several AOP mechanisms.

1. INTRODUCTION

Current AOP models apply a bottom-up abstractions. In a bottom-up abstraction, a model is built by identifying and generalizing common patterns. Since each AOP instance is useful precisely because it does some cross-cutting thing very different, an abstraction over several AOP instances often results in a fine-grained model. Such models would likely need to be further extended to fit future instances of AOP. These AOP models do not lend themselves towards designing new AOP languages.

Today the similarities and differences between existing AOP approaches are poorly understood. For example, the difference between AspectJ and Hyper/J [12] is sometimes explained using the syntactic property of symmetry. Under this view, AspectJ is asymmetric: an AspectJ program consists of a base Java program and a set of aspect definitions. Hyper/J is said to be symmetric: all concerns are written in Java.

Unfortunately, this view doesn't provide much insight into the essential difference between the two languages. It fails to explain how Hyper/J differs from a "symmetric" AspectJ-like implementation (e.g., AspectWerkz [1] or Classpects [10]).

Currently, weaving is understood from a base↔aspect perspective [8, 13, 4, 6, 14, 9]: an aspect concern is woven into the base program. This perspective is aligned with AspectJ's syntax. However, it doesn't explain well AOP languages that unify aspect and base abstractions (i.e., Hyper/J, AspectWerkz, Classpects). These languages do not distinguish between "aspect" and "base" concerns. They allow to weave "base" program elements into "aspect" concerns.

Even worse, the perspective doesn't match AOP languages where integration rules are specified separately from concern modules: the base↔aspect model presented in [8] was extended with a special $META$ element just to match Hyper/J.

We suggest a different perspective: a weaver composes output program elements by combining concern elements of an input AOP program. We call this perspective aop↔composed. Unlike base↔aspect, the aop↔composed view doesn't depend on a particular AOP language syntax. It reflects three essential observations:

1. the input program simply specifies a set of separate concern elements. No difference is made for "base" and "aspect" concerns.
2. an element of the output program/computation is combined from a subset of input concern elements
3. weaving logic is specified by the AOP integration rules

Using aop↔composed we model an abstract weaving process. The process model defines weaving functionality over sets of concern elements and composed program elements. It generalizes uniformly over existing AOP weavers, including AspectJ and Hyper/J mechanisms. We show that the weaving process specification leads to an abstract AOP mechanism model. We also discuss how it supports composition of multiple AOP mechanisms.

2. THE WEAVING PROCESS

In our approach, the weaving process integrates a concern specification modularized in the source AOP program into a composed program. More specifically, we represent the concern specification p as a set of n concern elements, $a_j \in \mathcal{A}$, $j = 1, \ldots, n$:

$$p = \{a_1, \ldots, a_n\}$$

and the composed program $\wp$ as an ordered set of m program elements, $x_i \in \mathcal{X}, i = 1, \ldots, m$:

$$\wp = \{x_1, \ldots, x_m\}$$

where $\mathcal{A}$ and $\mathcal{X}$ are domains of concern elements and program elements, respectively. Advice declarations in AspectJ exemplify concern elements. Computations are common program elements.

In order to understand how p is woven into $\wp$ we analyze the process of a program composition (Figure 1). The program is composed, starting from an empty program, in m transitions:

$$\phi = \wp_0 \overset{\tau_1}{\Rightarrow} \wp_1 \overset{\tau_2}{\Rightarrow} \ldots \overset{\tau_m}{\Rightarrow} \wp_m = \wp$$

where

$$\forall 1 \leq i \leq m : \wp_i = \{x_1, \ldots, x_i\}$$

Note, that this process maps well to both program AST construction and a program execution. In the first case, a transitions produces a new AST node and adds it to the tree. In the second case, the transition produces a computation and adds it to the computation tree.

Each transition $\tau_i, 1 \leq i \leq m$ takes two steps (Figure 2):

$$\wp_{i-1} \overset{\delta_i}{\Rightarrow} \wp_i^{raw} \overset{\omega_i}{\Rightarrow} \wp_i$$

where $\wp_i^{raw} = \{x_i, \ldots, x_{i-1}, x_i^{raw}\}$

The first step does not involve weaving and is external to the weaver. We call a mechanism that realizes this step a *default* mechanism. A programming language interpreter exemplifies the default mechanism for dynamic weavers. AST traversal algorithm (i.e., DFS or BFS) exemplifies the default mechanism for static weavers.

At each transition i the default mechanism supplies a function $\delta_i : \mathcal{X}^* \rightarrow \mathcal{X}$. This function produces a *raw* program element $x_i^{raw} \in \mathcal{X}$:

$$x_i^{raw} = \delta_i \ \wp_{i-1}$$

For example, the first computation produced by the interpreter while evaluating $\wp_{i-1}$ would be a raw computation. A node discovered by the AST traversal algorithm is a raw node.

At the second step of the transition, the AOP mechanism supplies a weaving function $\omega_i : \mathcal{X} \rightarrow \mathcal{X}$. The weaving function produces x_i by transforming x_i^{raw}:

$$x_i = \omega_i \ x_i^{raw}$$

We model ω_i functionality using two functions, namely $match_i$ and mix_i (Figure 3). The first function selects concern elements to be woven:

$$match_i : \mathcal{A}^* \rightarrow \mathcal{A}^*$$

The second function composes x_i by mixing the selected concern elements with x^{raw}:

$$mix_i : \mathcal{A}^* \rightarrow \mathcal{X} \rightarrow \mathcal{X}$$

such that

$$\omega_i = mix_i \ (match_i \ p)$$

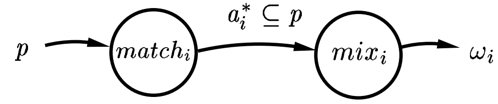

Figure 3: Weaving Function Model

In other words, at every transition τ_i the mechanism supplies $match_i$ and mix_i functions that combine concern elements with a raw program element x_i^{raw}.

2.1 Mapping to existing mechanisms

Mapping the process model to pointcut and advice mechanism (PA) of AspectJ is straightforward: (i) $\mathcal{X}$ is a domains of AspectJ computations; (ii) $\mathcal{A}$ denotes advice computations; (iii) join points identify composed program elements; (iv) $match$ functions are specified by pointcut designators (PCDs) and pointcut-to-advice mappings; (v) mix functions are specified via advice types (`before`, `after`, `around`); Other pointcut-and-advice mechanisms [6, 10, 1] (both "symmetric" and "assymetric") can be mapped to the model in the same manner.

The Hyper/J mechanism (CMP) composes a tree of *hypermodule nodes*[1] (composed program) from a tree of *hyperspace units* (concern code). At each weaving step CMP produces a set of child nodes for an empty[2] parent node. The weaving steps are specified by Hyper/J composition rules. Each rule contains a set of unit names (an association set), and a composition strategy. The strategy specifies, how referred units should be composed in the hypermodule.

In the model terms, (i) composed sets of nodes are $\mathcal{X}$ elements; (ii) $\mathcal{A}$ elements are units; (iii) $match$ functions are specified by association sets; (iv) mix functionality is given by composition strategies The default mechanism is given by a node tree traversal algorithm.

The Demeter [7] mechanism (TRV) allows to define structure-shy concerns. The concerns can *adapt* to a changes in a base program structure. Adaptation logic is specified by Demeter traversals. The traversals, however, does not specify how adapted concerns integrated (i.e., weaved) into the composed program. The integration happens in a regular manner by calling the traversals explicitly from the base program.

The Demeter addresses a concern *adaptation* rather than concern *integration* problem. TRV is not a weaver (at least in the same sense as AspectJ and Hyper/J mechanisms). It implements different process that cannot be expressed in terms of the weaving process model.

3. TOWARD A CONSTRUCTIVE MODEL

In general, an AOP language allows to specify a wide range of composition processes. The language supplies means of expressing mix and match logic for each process transition. An AOP mechanism then:

1. identifies a transition

[1]In Hyper/J terminology the hypermodule nodes are referred as hypermodule units. We change the term to avoid confusion with hyperspace units.

[2]A node with empty list of children.

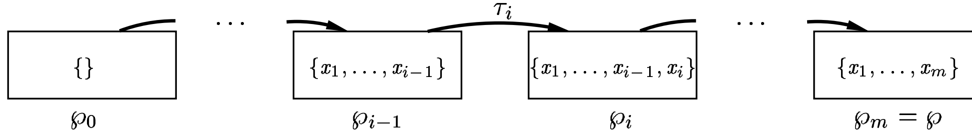

Figure 1: Program Composition Process

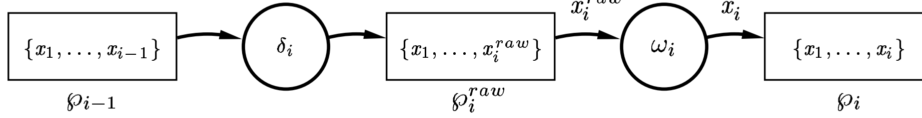

Figure 2: Weaving Process

2. searches a source program for applicable matching rules, and constructs a $match$ function

3. selects appropriate mixing rules from the program, and constructs a mix function

4. constructs weaving function ω

Further elaboration leads to an abstract AOP mechanism model that is both general and constructive. The model generalizes over specific mechanisms by realizing the abstract weaving process. It guides mechanism design by specifying essential weaver elements and processes.

4. TOWARD A COMPOSITIONAL MODEL

The problem of third-party AOP mechanism composition is the problem of integrating third-party mechanisms with a base language semantics. The weaving process model abstracts the base semantics as a default mechanism. For example, Java expression evaluation semantics is the default mechanism for the PA machinery.

Semantics of the default mechanism specifies a set of mappings. In terms of the process models, these mappings define program composition transitions. An AOP mechanism integration introduces weaving steps to the default mechanism by *transforming* the mappings. For example, PA mechanism can be combined with Java by transforming (a subset of) Java expression evaluation reductions.

Multiple AOP mechanisms are connected by composing their weaving functions *at every transition*.[3] For example, for n AOP mechanisms the n-mechanism weaving function ω_i^N can be constructed as:

$$\omega_i^N = \omega_i^1 \, \omega_i^2 \, \ldots \, \omega_i^n$$

where $\omega_i^j, 1 \leq j \leq n$ is a weaving function produced by j^{th} AOP mechanism at i^{th} transition.

5. RELATED WORK

Related work falls into two general categories, namely AOP models and multi-mechanism composition.

[3] Assuming that all the mechanisms transform the same type of program elements.

5.1 Modeling AOP

Existing models of AOP have focused either on a specific kind of AOP weavers, or on a bottom-up generalization of several AOP instances.

Models in the first category mostly adopt pointcut and advice perspective on AOP, focusing on dynamic advice weaving. Some of them give semantics to specific AspectJ-like mechanisms [14, 9, 6]. The others explain pointcut and advice weavers more generally. For instance, Walker et al. [13] defined aspects through explicitly labeled program points and first-class dynamic advice. Jagadeesan et al. [4] used pointcut and advice to define AOP functionality. It is not clear that these approaches can help in constructing AOP weavers of different kinds (e.g., Hyper/J-like mechanism).

Masuhara and Kiczales [8] identify and generalize common patterns in AspectJ, Demeter, and Hyper/J mechanisms. The generalization resulted in a fine-grained model that does not abstract neatly over all the mechanisms. For example, one of model elements ($META$) relates exclusively to the Hyper/J mechanism, not to the others. This model would likely need to be further extended to fit other instances of AOP.

Overall, existing AOP models are built bottom-up by generalizing over existing AOP instances or specific AOP functionalities. These models do not seem to generalize over all weaver mechanisms, and do not generally lend themselves towards constructing new weaver-based AOP mechanisms.

5.2 Composing multiple AOP mechanisms

Lieberherr et al. [11] present a framework for aspect compilation that allows to combine multiple domain-specific AOP extensions. The framework's composition semantics is to reduce all extensions to a single general-purpose AOP language (AspectJ). Specifically, given a set of programs written in different extensions, XAspects produces a single program in AspectJ.

Unfortunately, only a subset of AOP extensions is expressible in AspectJ. Hence, XAspects doesn't achieve composition in general. The approach to mechanism composition and collaboration we seek doesn't involve an AOP language assembler. It can combine arbitrary weaver-based AOP mechanisms. Comparing to XAspects our solution is more general.

6. CONCLUSION

The paper presents a model of an abstract weaving process. The model defines a weaver as a program element transformer, a special

case of a composed program transformer.

The absence of a so-called base program distinguishes our approach from conventional weaver models where weaving combines concern code with the base program.

In terms of the XF framework, we would define the weaver as:

$$A \to X \to X$$

where A is a domain of aspect programs, and X is a domain of composition results.

Absence of a base program is unique and crucial feature of our approach. The so-called base program is given implicitly, and is not actually required at all. Normally, if a base program exists, it supplies the first raw program element. For example, the *main* method call computation that starts the program execution in AspectJ is always drawn from the base Java program. However, program elements transformed by the weaver are generally different from the base program. As a result, at each transition the concern is integrated with a current composed program, *not* with the base program.

Our perspective contributes to understanding the weaving process, designing new weaver mechanisms, and explaining existing weavers in a uniform fashion. Moreover, we show that our approach supports composition of multiple AOP mechanisms.

7. REFERENCES

[1] J. Bonêr. Aspectwerkz - dynamic aop for java. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, Mancaster, UK, 2004. AOSD 2004, ACM Press.

[2] L. Cardelli, editor. *Proceedings of the 17th European Conference on Object-Oriented Programming*, number 2743 in Lecture Notes in Computer Science, Darmstadt, Germany, July21-25 2003. ECOOP 2003, Springer Verlag.

[3] A. Colyer. Aspectj. pages 123–143. Addison-Wesley, Boston, 2005.

[4] R. Jagadeesan, A. Jeffrey, and J. Riely. An untyped calculus for aspect oriented programs. In Cardelli [2], pages 54–73.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. ECOOP 2001, Springer Verlag.

[6] R. Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 41–55, Enschede, The Netherlands, Apr. 2002. AOSD 2002, ACM Press.

[7] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS-Kent Publishing, 1996.

[8] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In Cardelli [2], pages 2–28.

[9] H. Masuharaa, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In AOSD 2002 FOAL Workshop.

[10] H. Rajan and K. Sullivan. Classpects: Unifying aspect- and object-oriented program design. In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, USA, May 15-21 2005. ICSE 2005, ACM Press.

[11] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain specific aspect languages. In *Companion of the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 28–37, Anaheim, California, 2003. ACM Press.

[12] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In D. Garland and J. Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, Los Angeles, California, May 16-22 1999. ICSE 1999, IEEE Computer Society.

[13] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 127–139, Uppsala, Sweden, Aug. 2003. ACM Press.

[14] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Prog. Lang. Syst.*, 26(5):890—910, Sept. 2004.